M&G No. 60001.0264US01/MS303918.1

# PROGRAMMABLE OBJECT MODEL FOR EXTENSIBLE MARKUP LANGUAGE MARKUP IN AN APPLICATION

## Related Applications

United States Utility Patent Application by applicant matter number 60001.0263US01/MS303917.1, entitled "Programmable Object Model for Extensible Markup Language Schema Validation," and United States Utility Patent Application by applicant matter number 60001.0270US01/MS303919.1, entitled "Programmable Object Model for Namespace or Schema Library Support in a Software Application," are hereby incorporated by reference.

## Copyright Notice



## Field of the Invention

The present invention relates to programmable object models. More particularly, the present invention relates to a programmable object model for Extensible Markup Language markup in a software application.

## Background of the Invention

Computer software applications allow users to create a variety of documents to assist them in work, education, and leisure. For example, popular word processing applications allow users to create letters, articles, books, memoranda, and the like. Spreadsheet applications allow users to store, manipulate, print, and display a

variety of alphanumeric data. Such applications have a number of well known strengths including rich editing, formatting, printing, calculation and on-line and off-line editing.

Most computer software applications do not contain all necessary programming for providing functionality required or desired by every potential user. Many programmers often wish to take advantage of an existing application's capabilities in their own programs or to customize the functionality of an application and make it more suitable for a specific set of users or actions. For example, a programmer working in the financial industry may wish to customize a word processor for a user audience consisting of financial analysts editing financial reports. In recent years, the Extensible Markup Language has been used widely as an interchangeable data format for many users. A word processor or other application capable of receiving and editing XML data provides great value to its users. However, because the native XML functionality of such an application is not exposed to users/programmers wishing to customize the functionality to further take advantage of the functionality for their own programs, such users/programmers are limited in their ability to utilize the XML functionality.

Accordingly, there is a need in the art for a programmable object model for allowing users/programmers to programmatically access the XML functionality of a software application in order to customize or otherwise modify the XML functionality to enhance their utilization of the functionality. It is with respect to these and other considerations that the present invention has been made.

## Summary of the Invention

The present invention provides methods and systems for allowing a user to programmatically access the Extensible Markup Language (XML) functionality of a software application by providing a programmable object model. According to one aspect of the present invention, a programmable object model comprised of a plurality of object-oriented message calls or application programming interfaces is provided for allowing a user to access the XML functionality of an application by sending one or

more object-oriented message calls or application programming interfaces to the XML functionality of a given application along with any required parameters for customizing or otherwise manipulating XML markup applied to a document. Once the user has access to the XML functionality of a given application, the user may insert or delete XML markup in arbitrary locations in the document, the user may access richly formatted contents in areas contained inside an XML element or node, the user may locate specific XML nodes or elements in a document using queries, and the user may control or otherwise manipulate the application's XML-related settings and properties.

These and other features, advantages, and aspects of the present invention may be more clearly understood and appreciated from a review of the following detailed description of the disclosed embodiments and by reference to the appended drawings and claims.

## Brief Description of the Drawings

Fig. 1 is a simplified block diagram of a computing system and associated peripherals and network devices that provide an exemplary operating environment for the present invention.

Fig. 2 is a simplified block diagram illustrating interaction between software objects according to an object-oriented programming model.

Fig. 3 is a block diagram illustrating interaction between a document, an attached schema file, and a schema validation functionality model.

Fig. 4 is a block diagram illustrating interaction between a third party application and the XML functionality of a software application.

## Detailed Description of the Preferred Embodiment

Embodiments of the present invention are directed to methods and systems for allowing a user to programmatically call the Extensible Markup Language (XML) functionality of a software application for customizing, utilizing, and otherwise manipulating objects and properties of the XML functionality and for customizing,

manipulating and enhancing the utilization of XML markup in a document. These embodiments may be combined, other embodiments may be utilized, and structural changes may be made without departing from the spirit or scope of the present invention. The following detailed description is therefore not to be taken in a limiting senses and the scope of the present invention is defined by the appended claims and their equivalents.

Referring now to the drawings, in which like numerals represent like elements through the several figures, aspects of the present invention and the exemplary operating environment will be described. Fig. 1 and the following discussion are intended to provide a brief, general description of a suitable computing environment in which the invention may be implemented. While the invention will be described in the general context of program modules that execute in conjunction with an application program that runs on an operating system on a personal computer, those skilled in the art will recognize that the invention may also be implemented in combination with other program modules.

Generally, program modules include routines, programs, components, data structures, and other types of structures that perform particular tasks or implement particular abstract data types. Moreover, those skilled in the art will appreciate that the invention may be practiced with other computer system configurations, including hand-held devices, multiprocessor systems, microprocessor-based or programmable consumer electronics, minicomputers, mainframe computers, and the like. The invention may also be practiced in distributed computing environments where tasks are performed by remote processing devices that are linked through a communications network. In a distributed computing environment, program modules may be located in both local and remote memory storage devices.

Turning now to Fig. 1, an illustrative computer architecture for a personal computer 2 for practicing the various embodiments of the invention will be described. The computer architecture shown in Fig. 1 illustrates a conventional personal computer, including a central processing unit 4 ("CPU"), a system memory 6, including a random access memory 8 ("RAM") and a read-only memory ("ROM") 10,

and a system bus 12 that couples the memory to the CPU 4. A basic input/output system containing the basic routines that help to transfer information between elements within the computer, such as during startup, is stored in the ROM 10. The personal computer 2 further includes a mass storage device 14 for storing an operating system 16, application programs, such as the application program 305, and data.

The mass storage device 14 is connected to the CPU 4 through a mass storage controller (not shown) connected to the bus 12. The mass storage device 14 and its associated computer-readable media, provide non-volatile storage for the personal computer 2. Although the description of computer-readable media contained herein refers to a mass storage device, such as a hard disk or CD-ROM drive, it should be appreciated by those skilled in the art that computer-readable media can be any available media that can be accessed by the personal computer 2.

By way of example, and not limitation, computer-readable media may comprise computer storage media and communication media. Computer storage media includes volatile and non-volatile, removable and non-removable media implemented in any method or technology for storage of information such as computer-readable instructions, data structures, program modules or other data. Computer storage media includes, but is not limited to, RAM, ROM, EPROM, EEPROM, flash memory or other solid state memory technology, CD-ROM, DVD, or other optical storage, magnetic cassettes, magnetic tape, magnetic disk storage or other magnetic storage devices, or any other medium which can be used to store the desired information and which can be accessed by the computer.

According to various embodiments of the invention, the personal computer 2 may operate in a networked environment using logical connections to remote computers through a TCP/IP network 18, such as the Internet. The personal computer 2 may connect to the TCP/IP network 18 through a network interface unit 20 connected to the bus 12. It should be appreciated that the network interface unit 20 may also be utilized to connect to other types of networks and remote computer systems. The personal computer 2 may also include an input/output controller 22 for receiving and processing input from a number of devices, including a keyboard or mouse (not

shown). Similarly, an input/output controller 22 may provide output to a display screen, a printer, or other type of output device.

As mentioned briefly above, a number of program modules and data files may be stored in the mass storage device 14 and RAM 8 of the personal computer 2, including an operating system 16 suitable for controlling the operation of a networked personal computer, such as the WINDOWS XP operating system from MICROSOFT CORPORATION of Redmond, Washington. The mass storage device 14 and RAM 8 may also store one or more application programs. In particular, the mass storage device 14 and RAM 8 may store an application program 305 for creating and editing an electronic document 310. For instance, the application program 305 may comprise a word processing application program, a spreadsheet application, a contact application, and the like. Application programs for creating and editing other types of electronic documents may also be used with the various embodiments of the present invention. A schema file 330and a namespace/schema library 400, described below, are also shown.

Exemplary embodiments of the present invention are implemented by communications between different software objects in an object-oriented programming environment. For purposes of the following description of embodiments of the present invention, it is useful to briefly to describe components of an object-oriented programming environment. Fig. 2 is a simplified block diagram illustrating interaction between software objects according to an object-oriented programming model. According to an object-oriented programming environment, a first object 210 may include software code, executable methods, properties, and parameters. Similarly, a second object 220 may also include software code, executable methods, properties, and parameters.

A first object 210 may communicate with a second object 220 to obtain information or functionality from the second object 220 by calling the second object 220 via a message call 230. As is well know to those skilled in the art of object-oriented programming environment, the first object 210 may communicate with the second object 220 via application programming interfaces (API) that allow two disparate software objects 210, 220 to communicate with each other in order to obtain

information and functionality from each other. For example, if the first object 210 requires the functionality provided by a method contained in the second object 220, the first object 210 may pass a message call 230 to the second object 220 in which the first object identifies the required method and in which the first object passes any required parameters to the second object required by the second object for operating the identified method. Once the second object 220 receives the call from the first object, the second object executes the called method based on the provided parameters and sends a return message 250 containing a value obtained from the executed method back to the first object 210.

For example, in terms of embodiments of the present invention, and as will be described below, a first object 210 may be a third party customized application that passes a message to a second object such as an Extensible Markup Language schema validation object whereby the first object identifies a method requiring the validation of a specified XML element in a document where the specified XML element is a parameter passed by the first object with the identified method. Upon receipt of the call from the first object, according to this example, the schema validation object executes the identified method on the specified XML element and returns a message to the first object in the form of a result or value associated with the validated XML element. Operation of object-oriented programming environments, as briefly described above, are well known to those skilled in the art.

As described below, embodiments of the present invention are implemented through the interaction of software objects in the use, customization, and application of components of the Extensible Markup Language (XML). Fig. 3 is a block diagram illustrating interaction between a document, an attached schema file, and a schema validation functionality module. As is well known to those skilled in the art, the Extensible Markup Language (XML) provides a method of describing text and data in a document by allowing a user to create tag names that are applied to text or data in a document that in turn define the text or data to which associated tags are applied. For example referring to Fig. 3, the document 310 created with the application 305 contains text that has been marked up with XML tags 315, 320, 325. For example, the text

"Greetings" is annotated with the XML tag <title>. The text "My name is Sarah" is annotated with the <body> tag. According to XML, the creator of the <title> and <body> tags is free to create her own tags for describing the tags to which those tags will be applied. Then, so long as any downstream consuming application or computing machine is provided instructions as to the definition of the tags applied to the text, that application or computing machine may utilize the data in accordance with the tags. For example, if a downstream application has been programmed to extract text defined as titles of articles or publications processed by that application, the application may parse the document 310 and extract the text "Greetings," as illustrated in Fig. 3 because that text is annotated with the tag <title>. The creator of the particular XML tag naming for the document 310, illustrated in Fig. 3, provides useful description for text or data contained in the document 310 that may be utilized by third parties so long as those third parties are provided with the definitions associated with tags applied to the text or data.

According to embodiments of the present invention, the text and XML markup entered into the document 310 may be saved according to a variety of different file formats and according to the native programming language of the application 305 with which the document 310 is created. For example, the text and XML markup may be saved according to a word processing application, a spreadsheet application, and the like. Alternatively, the text and XML markup entered into the document 310 may be saved as an XML format whereby the text or data, any applied XML markup, and any formatting such as font, style, paragraph structure, etc. may be saved as an XML representation. Accordingly, downstream or third party applications capable of understanding data saved as XML may open and consume the text or data thus saved as an XML representation. For a detailed discussion of saving text and XML markup and associated formatting and other attributes of a document 310 as XML, see U.S. Patent Application entitled "Word Processing Document Stored in a Single XML File that may be Manipulated by Applications that Understanding XML," U.S. Serial No. 10/187,060, filed June 28, 2002, which is incorporated herein by reference as if fully set out herein.

In order to provide a definitional framework for XML markup elements (tags) applied to text or data, as illustrated in Fig. 3, XML schema files are created which contain information necessary for allowing users and consumers of marked up and stored data to understand the XML tagging definitions designed by the creator of the document. Each schema file also referred to in the art as a Namespace or XSD file preferably includes a listing of all XML elements (tags) that may be applied to a document according to a given schema file. For example, a schema file 330, illustrated in Fig. 3, may be a schema file containing definitions of certain XML elements that may be applied to a document 310 including attributes of XML elements or limitations and/or rules associated with text or data that may be annotated with XML elements according to the schema file. For example, referring to the schema file 330 illustrated in Fig. 3, the schema file is identified by the Namespace "intro" the schema file includes a root element of <intro card>.

According to the schema file 330, the <intro card> element serves as a root element for the schema file and also as a parent element to two child elements <title> and <body>. As is well known to those skilled in the art, a number of parent elements may be defined under a single root element, and a number of child elements may be defined under each parent element. Typically, however, a given schema file 330 contains only one root element. Referring still to Fig. 3, the schema file 330 also contains attributes 340 and 345 to the <title> and <body> elements, respectfully. The attributes 340 and 345 may provide further definition or rules associated with applying the respective elements to text or data in the document 310. For example, the attribute 345 defines that text annotated with the <title> element must be less than or equal to twenty-five characters in length. Accordingly, if text exceeding twenty-five characters in length is annotated with the <title> element or tag, the attempted annotation of that text will be invalid according to the definitions contained in the schema file 330.

By applying such definitions or rules as attributes to XML elements, the creator of the schema may dictate the structure of data contained in a document associated with a given schema file. For example, if the creator of a schema file 330 for defining XML markup applied to a resume document desires that the experience section

of the resume document contain no more than four present or previous job entries, the creator of the schema file 330 may define an attribute of an <experience> element, for example, to allow that no more than four present or past job entries may be entered between the <experience> tags in order for the experience text to be valid according to the schema file 330. As is well known to those skilled in the art, the schema file 330 may be attached to or otherwise associated with a given document 310 for application of allowable XML markup defined in the attached schema file to the document 310. According to one embodiment, the document 310 marked up with XML elements of the attached or associated schema file 330 may point to the attached or associated schema file by pointing to a uniform resource identifier (URI) associated with a Namespace identifying the attached or associated schema file 330.

According to embodiments of the present invention, a document 310 may have a plurality of attached schema files. That is, a creator of the document 310 may associate or attach more than one schema file 330 to the document 310 in order to provide a framework for the annotation of XML markup from more than one schema file. For example, a document 310 may contain text or data associated with financial data. A creator of the document 310 may wish to associate XML schema files 330 containing XML markup and definitions associated with multiple financial institutions. Accordingly, the creator of the document 310 may associate an XML schema file 330 from one or more financial institutions with the document 310. Likewise, a given XML schema file 330 may be associated with a particular document structure such as a template for placing financial data into a desirable format.

According to embodiments of the present invention, a collection of XML schema files and associated document solutions may be maintained in a Namespace or schema library located separately from the document 310. The document 310 may in turn contain pointers to URIs in the Namespace or schema library associated with the one or more schema files attached to otherwise associated with the document 310. As the document 310 requires information from one or more associated schema files, the document 310 points to the Namespace or schema library to obtain the required schema definitions. For a detailed description of the use of an operation of Namespace or

schema libraries, see U.S. Patent Application entitled "System and Method for Providing Namespace Related Information," U.S. Serial No. 10/184,190, filed June 27, 2002, and U.S. Patent Application entitled "System and Method for Obtaining and Using Namespace Related Information for Opening XML Documents," U.S. Serial No. 10/185,940, filed June 27, 2002, both U.S. patent applications of which are incorporated herein by reference as if fully set out herein. For a detailed description of a mechanism for downloading software components such as XML schema files and associated solutions from a Namespace or schema library, see US Patent Application entitled Mechanism for Downloading Software Components from a Remote Source for Use by a Local Software Application, US Serial No. 10/164,260, filed June 5, 2002.

Referring still to Fig. 3, a schema validation functionality module 350 is illustrated for validating XML markup applied to a document 310 against an XML schema file 330 attached to or otherwise associated with the document 310, as described above. As described above, the schema file 330 sets out acceptable XML elements and associated attributes and defines rules for the valid annotation of the document 310 with XML markup from an associated schema file 330. For example, as shown in the schema file 330, two child elements <title> and <body> are defined under the root or parent element <intro card>. Attributes 340, 345 defining the acceptable string length of text associated with the child elements <title> and <body> are also illustrated. As described above, if a user attempts to annotate the document 310 with XML markup from a schema file 330 attached to or associated with the document in violation of the XML markup definitions contained in the schema file 330, an invalidity or error state will be presented. For example, if the user attempts to enter a title string exceeding twenty-five characters, that text entry will violate the maximum character length attribute of the <title> element of the schema file 330. In order to validate XML markup applied to a document 310, against an associated schema file 330, a schema validation module 350 is utilized. As should be understood by those skilled in the art, the schema validation module 350 is a software module including computer executable instructions sufficient for comparing XML markup and associated text entered in to a

document 310 against an associated or attached XML schema file 330 as the XML markup and associated text is entered in to the document 310.

According to embodiments of the present invention, the schema validation module 350 compares each XML markup element and associated text or data applied to the document 310 against the attached or associated schema file 330 to determine whether each element and associated text or data complies with the rules and definitions set out by the attached schema file 330. For example, if a user attempts to enter a character string exceeding twenty-five characters annotated by the <title> elements 320, the schema validation module will compare that text string against the text string attribute 340 of the attached schema file 330 and determine that the text string entered by the user exceeds the maximum allowable text string length. Accordingly, an error message or dialogue will be presented to the user to alert the user that the text string being entered by the user exceeds the maximum allowable character length according to the attached schema file 330. Likewise, if the user attempts to add an XML markup element between the <title> and the <body> elements, the schema validation module 350 will determine that the XML markup element applied by the user is not a valid element allowed between the <title> and <body> elements according to the attached schema file 330. Accordingly, the schema validation module 350 will generate an error message or dialogue to the user to alert the user of the invalid XML markup.

Programmable Object Model for XML Markup in an Application

As described above with reference to Fig. 3, an application 305 may allow a user to apply Extensible Markup Language (XML) elements to a document 310. Further, a schema file 330 may be attached to or associated with the document 310 for providing a framework of definitions and rules for applying XML markup elements to the document 310 in accordance with a schema file 330. According to embodiments of the present invention, a user is allowed to programmatically call the XML functionality of an application 305 for gaining access to XML markup applied to a document and for editing XML markup applied to the document. Further, the user may call the XML

functionality of the application 305 for setting and adding attributes on XML elements, and adding elements, in spite of an attached or associated schema file 330. Access to the XML functionality of the application 305 also allows a user to access richly formatted text or data contents contained within an XML node or to find specific XML nodes (elements) in the document 310 by using one or more XML XPath queries.

According to embodiments of the present invention, a user may access the XML functionality of an application by sending one or more object-oriented message calls or application programming interfaces to the XML functionality of the application 305 as described above with reference to Fig. 2. By exposing the user to the XML functionality of the application 305, the user may programmatically call the XML functionality of the application 305, directly through use of provided message calls or API's or from a third party software application capable of communicating with the XML functionality of the application 305 via object-oriented message calls or API's. The user may prepare a third party application according to a variety of different programming languages such as C, C++, C#, Visual Basic, and the like.

Fig. 4 is a block diagram illustrating interaction between a third party application and the XML functionality of an application 305. Referring to the block diagram illustrated in Fig. 4, an application 305 is illustrated having a document 310 in which has been entered XML markup elements 315, 320, 325 and associated text. To the right of the document 310 is an XML markup pane 405 for providing the user a tree-structured outline including an identification of a schema file or Namespace 410 attached to the document 310 and showing the root element or parent element 420 and associated child elements 430, 440, 450, 460, 480. An attribute 470 is shown associated with the child element 460. As is well known to those skilled in the art, and as described above with reference to Fig. 3, the user operating the software application 305 may apply XML elements from an associated or attached XML schema file to the document 310, as illustrated in Fig. 4.

According to embodiments of the present invention, and as described above, the user may programmatically access the XML functionality of the software application to customize or otherwise manipulate the application of XML elements

from the associated or attached schema file 330 to the document 310. For example, the user may desire to programmatically insert additional XML markup into the document at a specified range by sending an object-oriented message call specifying a method for inserting a given XML markup element into a particular range within the document. For another example, the user may desire to apply an Extensible Stylesheet Language Transformation (XSLT) to the document 310 in order to transform the document 310 to a format required by the user, for example HTML format. In order to effect such a transformation, the user is provided an object-oriented message call for calling the XML functionality of the application 305 and for providing the location of a desired XSLT transformation file for applying to the document 310.

The following is a description of objects and associated properties comprising object-oriented message calls (application programming interfaces) provided to a user for allowing the user to directly access the XML functionality of the software application 305, as described above. Following each of the objects or associated properties set out below is a description of the operation and functionality of the object or associated property.

**Application object**

The following are properties and methods of this object related to XML markup.

**.PrintXMLTag property**

A property controlling whether XML tags are printed out together with the contents of the document.

**Document object**

The following are properties and methods of this object related to XML markup.

**.TranformDocument**

A method to transform the XML document using an XSLT transformation and open its output. It can accept the following parameters.

*Path* – location to the XSLT transformation file.
*DataOnly* – a flag indicating whether what is passed to the transformation is the entire XML representation of the document (including native XML representations) or only the non-native XML markup embedded in the document.

**.XMLAfterInsert event**

An event firing right after a new XML element is inserted into the document by a user action. This event firing passes the following parameters to an event handler procedure.

*NewXMLNode* – a pointer to the new XML node object being inserted
*InUndoRedo* – a flag indicating whether the insertion occurred as a result of an Undo or Redo action in the application.

**.XMLBeforeDelete event**

An event firing right before an XML element is deleted from the document by a user action. This event firing passes the following parameters to an event handler procedure.

*DeletedRange* – an object pointing to the area of the document being affected by the deletion.
*OldXMLNode* – a pointer to the XML node being deleted.
*InUndoRedo* – a flag indicating whether the deletion occurred as a result of an Undo or Redo action in the application.

**.XMLHideNamespaces property**

A property controlling the appearance of the Namespace text when the element name shows up in the application's user interface.

**.XMLNodes property**

A read only property pointing to an XMLNodes collection representing all the XML nodes in the document.

**.XMLSaveDataOnly property**

A property controlling how the application saves the XML markup embedded in its document, including whether it is only that markup and the text contents, or whether it has other data that may be native to the application and created automatically.

**.XMLSaveThroughXSLT property**

A property controlling the location of the XSLT transform to be applied by the application to the document automatically upon saving the document so that only the output of the XSLT transformation is saved.

**.XMLUseXSLTWhenSaving property**

A property that controls whether an XSLT transform should automatically be applied to a document right before the document is saved so that only the results of that transformation are saved.

**View object**

The following are properties and methods of this object related to XML markup.

**.ShowXMLMarkup property**

A property that controls whether or not the XML markup embedded in the document is visible to the user as part of the text the user is editing in the document.

**XMLSchemaReferences collection object** - an object for providing access to XMLSchemaReference objects. This object represents the schemas attached to the document that the XML markup is based on and validated against. The following are methods and properties of the object that are relevant to the XML markup object model.

**.ShowPlaceholderText property**

A property controlling the automatic appearance of element names as visible placeholders in the document whenever the elements empty and when the tags themselves are not visible.

**Range object** - this is an object representing an arbitrary part of a document's content. The following are properties and methods of this object related to XML markup.

**.InsertXML() method**

A method for inserting XML markup into the document at the location represented by the Range object. It can accept the following parameters.

*XML* – the text string with the XML markup being inserted.
*Transform* – an XSLT transformation to be applied to the inserted markup right before it is inserted into the document.

**.XML property**

A read only property returning the XML representation of the area of the document represented by the Range object. This property can accept the following parameters.

*DataOnly* – a flag indicating that only the XML markup non-native to the application is to be returned.

**.XMLNodes property**

A read only property returning the pointer to the XMLNodes collection representing all the XML elements located in the area of the document represented by the Range object.

**.XMLParentNode property**

A read only property returning the innermost XML element in the document that contains the area represented by the Range object.

**Selection object** - an object representing the currently selected part of a document's content. The following are properties and methods of this object related to XML markup.

**.InsertXML() method**

A method for inserting XML markup into the document at the location represented by the Selection object. This method can accept the following parameters.

*XML* – the text string with the XML markup being inserted.
*Transform* – an XSLT transformation to be applied to the inserted markup right before it is inserted into the document.

**.XML property**

A read only property returning the XML representation of the area of the document represented by the Selection object. This property can accept the following parameters.

*DataOnly* – a flag indicating that only the XML markup non-native to the application is to be returned.

**.XMLNodes property**

A read only property returning the pointer to the XMLNodes collection representing all the XML elements located in the selected area of the document represented by the Selection object.

**.XMLParentNode property**

A read only property returning the innermost XML element in the document that contains the selected area represented by the Selection object.

**XMLNodes collection object** - an object representing a collection of XMLNode objects. The following are properties and methods of this object related to XML markup.

**.Add() method**

A method to create and add to the collection a new XMLNode object and apply it to the selected part of the document. The method can accept the following parameters.

*Name* – the name of the XML element to be created.

*Namespace* – the URI of that element determining which Namespace it is a member.

*Range* – a pointer to the part of the document to which this XML node is to be applied.

**.Application property**

A read only pointer to the application object representing the application of this object model.

**.Count property**

A read only property returning the number of XML nodes in the collection.

**.Creator property**

A read only pointer to the creator of the object.

**.Item() method**

A method for accessing the individual members of this collection using an numerical index or a search keyword. The method can accept the following parameters:

*Index* – a number representing the position of the requested XMLNode object in the Namespace library. The method can also be a text string representing the name of the requested node.

**.Parent property**

A read only property returning the parent object of the collection. This property returns a pointer to the document object from which the XMLNodes collection is accessed.

**XMLNode object** - an object representing an XML node in the document. The following are properties and methods of this object related to XML markup.

**.Application property**

A read only pointer to the application object representing the application of this object model.

**.Attributes property**

A read only property pointing to the XMLNodes collection consisting of all the attribute nodes associated with this XML node object.

**.BaseName property**

A property returning the name of this XML element without any prefixes.

**.ChildNodes property**

A property pointing to the XMLNodes collection that consists of all the elements that are child nodes of this element.

**.Copy() method**

A method to copy this XML element and all of its contents to the clipboard.

**.Creator property**

A read only pointer to the creator of the object.

**.Cut() method**

A method to copy this XML element and all of its contents to the clipboard and to remove that element and its contents from the document.

**.Delete() method**

A method to remove this XML element from the document without affecting its contents.

**.FirstChild property**

A property pointing to the first XML element that is a child of this XML element.

**.HasChildNodes property**

A property indicating whether the XML element represented by this object has any child nodes or is empty.

**.LastChild property**

A property pointing to the last XML element that is a child of this XML element.

**.Level property**

A property indicating whether the XML element represented by this object is at the inline level, the paragraph level, the table cell level, the table row level, the table level or any other special level supported by the application.

**.NamespaceURI property**

A property returning the URI of the Namespace of the XML element represented by this object.

**.NextSibling property**

A property returning a pointer to the XMLNode object that represents the next element following the element represented by this object.

**.NodeType property**

A property indicating the type of the XML node represented by this object. For example, it can determine whether the node is an element, or an attribute, or a text node or an annotation.

**.NodeValue property**

A property representing the text value to which the node represented by this object has been set.

**.OwnerDocument property**

A property returning a pointer to the document object in which this XML element is positioned.

**.Parent property**

A read only property returning the parent object of this object. This property returns a pointer to the collection of which the XMLNode object is a member.

**.ParentNode property**

A property returning a pointer to the object that represents the XML node of which this element is a child.

**.PlaceholderText property**

A property controlling the placeholder text that shows up in place of elements when those elements are empty and their tags are not visible.

**.PreviousSibling property**

A property returning a pointer to the XMLNode object that represents the previous element before the element represented by this object.

**.Range property**

A property returning a pointer to the Range object that represents the part of the document that the XML element contains. This provides a direct link between the XML markup in the document and the rich contents of the document, including data native to the application.

**.RemoveChild() method**

A method to remove an XML node that is a child of the node represented by this object. It can accept the following parameters.

*ChildNode* – a pointer to the XMLNode object that represents the child XML element to be removed.

**.SelectNodes() method**

A method to find all XML nodes in the document that match a specific query in the XPath language. It returns an XMLNodes collection consisting of all the XML nodes matching the query. It can accept the following parameters.

*XPath* – the xpath expression describing the query.

*PrefixMapping* – a mapping of prefixes to Namespaces used in the XPath expression.

*FastSearchSkippingTextNodes* – a performance optimization parameter that determines whether the text contents of XML elements should be

included in the search or if the search is only to be applied to the XML markup.

**.SelectSingleNode() method**

A method to find the first XML node in the document that matches a specific query in the XPath language. It returns an XMLNode object matching the query. It can accept the following parameters.

*XPath* – the xpath expression describing the query.

*PrefixMapping* – a mapping of prefixes to Namespaces used in the XPath expression.

*FastSearchSkippingTextNodes* – a performance optimization parameter that determines whether the text contents of XML elements should be included in the search or if the search is only to be applied to the XML markup.

**.SmartTag property**

A pointer to the SmartTag object associated with this XML element.

**.Text property**

A property returning the plain text contents of the XML element.

**.XML property**

A property returning the XML markup representation of the XML element and all of its contents. It can accept the following parameters.

*DataOnly* – a flag indicating that only the XML markup non-native to the application is to be returned.

As described herein, methods and systems are provided for allowing a user to programmatically access the Extensible Markup Language (XML) functionality of an application for accessing and customizing XML markup applied to a document created by the application and for controlling XML markup definitions and rules applied to the document by one or more associated or attached XML schema files. It will be apparent to those skilled in the art that various modifications or variations may be made in the present invention without departing from the scope or spirit of the invention. Other embodiments of the invention will be apparent to those skilled in the art from consideration of the specification and practice of the invention disclosed herein.